**Assignment 3.9**

Problem Statement:

● Explain the below concepts with an example in brief.

● Nosql Databases

● Types of Nosql Databases

● CAP Theorem

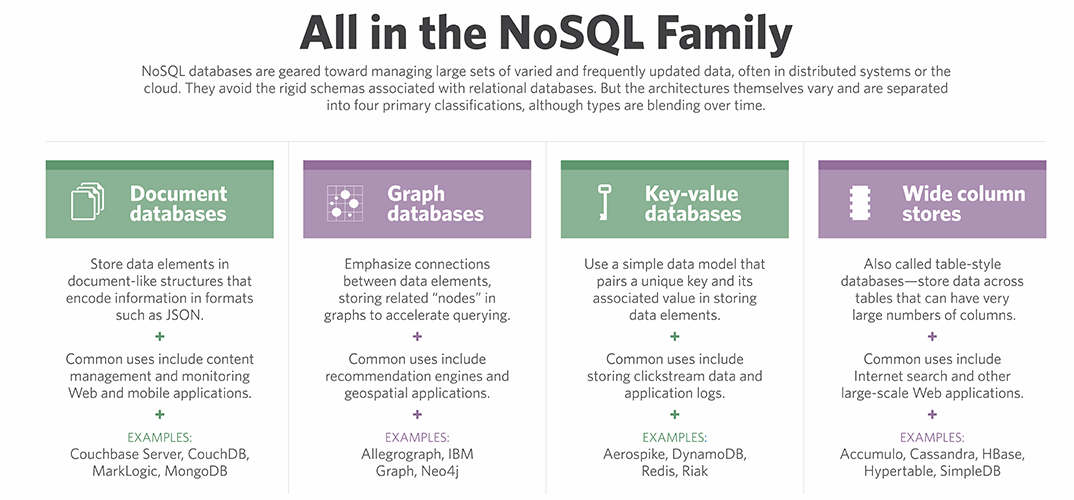
● HBase Architecture

● HBase vs RDBMS

**Nosql Databases**

NoSQL is an approach to databases that represents a shift away from traditional relational database management systems (RDBMS). To define NoSQL, it is helpful to start by describing SQL, which is a query language used by RDBMS. Relational databases rely on tables, columns, rows, or schemas to organize and retrieve data. In contrast, NoSQL databases do not rely on these structures and use more flexible data models. NoSQL can mean “not SQL” or “not only SQL.” As RDBMS have increasingly failed to meet the performance, scalability, and flexibility needs that next-generation, data-intensive applications require, NoSQL databases have been adopted by mainstream enterprises. NoSQL is particularly useful for storing unstructured data, which is growing far more rapidly than structured data and does not fit the relational schemas of RDBMS. Common types of unstructured data include: user and session data; chat, messaging, and log data; time series data such as IoT and device data; and large objects such as video and images.

**Types of Nosql Databases**



**Key-value stores**

Key-value stores, or key-value databases, implement a simple data model that pairs a unique key with an associated value. Because this model is simple, it can lead to the development of key-value databases, which are extremely performant and highly scalable for session management and caching in web applications. Implementations differ in the way they are oriented to work with RAM, solid-state drives or disk drives. Examples include Aerospike, Berkeley DB, MemchacheDB, Redis and Riak.

**Document databases**

Document databases, also called document stores, store semi-structured data and descriptions of that data in document format. They allow developers to create and update programs without needing to reference master schema. Use of document databases has increased along with use of JavaScript and the JavaScript Object Notation (JSON), a data interchange format that has gained wide currency among web application developers, although XML and other data formats can be used as well. Document databases are used for content management and mobile application data handling. Couchbase Server, CouchDB, DocumentDB, MarkLogic and MongoDB are examples of document databases.

**Wide-column stores**

Wide-column stores organize data tables as columns instead of as rows. Wide-column stores can be found both in SQL and NoSQL databases. Wide-column stores can query large data volumes faster than conventional relational databases. A wide-column data store can be used for recommendation engines, catalogs, fraud detection and other types of data processing. Google BigTable, Cassandra and HBase are examples of wide-column stores.

**Graph stores**

Graph data stores organize data as nodes, which are like records in a relational database, and edges, which represent connections between nodes. Because the graph system stores the relationship between nodes, it can support richer representations of data relationships. Also, unlike relational models reliant on strict schemas, the graph data model can evolve over time and use. Graph databases are applied in systems that must map relationships, such as reservation systems or customer relationship management. Examples of graph databases include AllegroGraph, IBM Graph, Neo4j and Tita

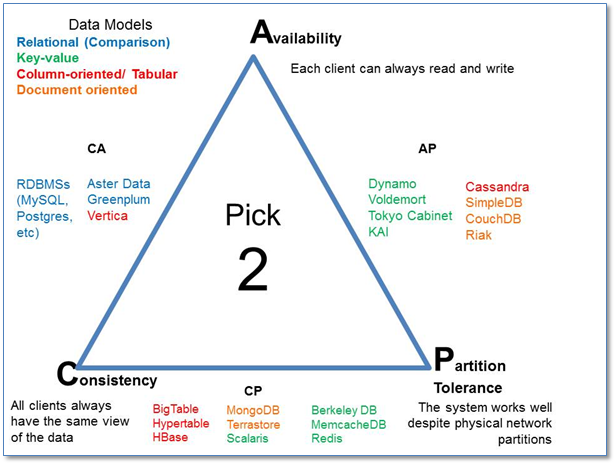
**CAP Theorem**

The BASE acronym was defined by Eric Brewer, who is also known for formulating the CAP theorem. Distributed systems are prone to outages and not safe from network failures. As system are being scaled up designing a resilient system had to deal with comlpexity incurred in the system. One of the [fallcy of the distributed system is that network are reliable](https://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing). This is where the CAP theorem comes into picture.

The CAP theorem, also known as Brewer's theorem, states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

* **Consistency:** all nodes see the same data at the same time.
* **Availability:** a guarantee that every request receives a response about whether it was successful or failed.
* **Partition tolerance:** the system continues to operate despite arbitrary message loss or failure of part of the system)

According to the theorem, a distributed system cannot satisfy all three of these guarantees at the same time.



Databases can be categorized as CP, AP or CA.

As mentioned earlier that network cannot be reliable. When a network failure happens to tolerate partition one has to choose one between **Consistency(CP)** or **Availability(AP)**.

* + CP - Consisteny/Parititon Tolerance - The most recent write is always visible to subsequent readers (single register linearizability). In the event of partitions, the system will block rather than return inconsistent data.
  + AP - Availability/Parititon Tolerance - Imagine having system running with multiple nodes. When parition occurs, the nodes will continue to server traffic and writes may not be replicated across all replicas and/or slaves. In such case during a read the system will return the most recent version of the data in presnt on a node, which could be stale. This system state will also accept writes that can be processed later when the partition is resolved.

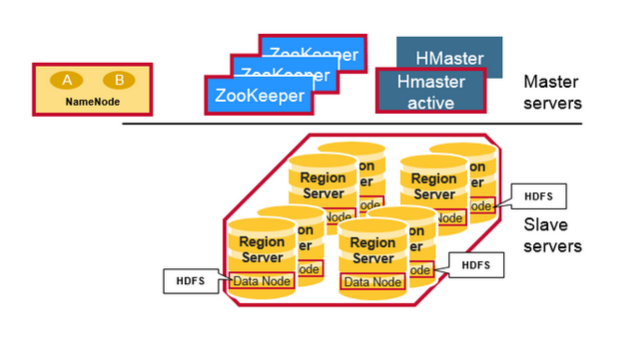
**HBase Architecture**

**HBase Architectural Components**

Physically, HBase is composed of three types of servers in a master slave type of architecture. Region servers serve data for reads and writes. When accessing data, clients communicate with HBase RegionServers directly. Region assignment, DDL (create, delete tables) operations are handled by the HBase Master process. Zookeeper, which is part of HDFS, maintains a live cluster state.

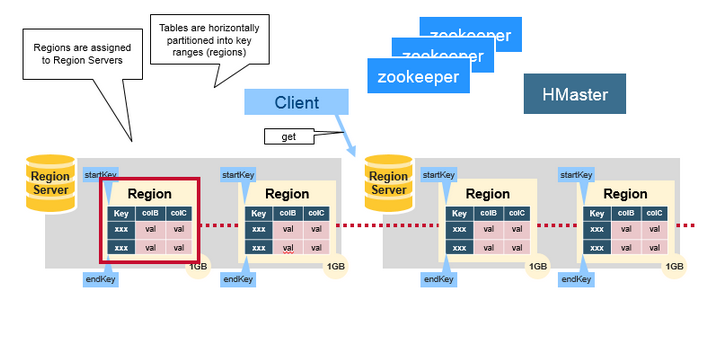
The Hadoop DataNode stores the data that the Region Server is managing. All HBase data is stored in HDFS files. Region Servers are collocated with the HDFS DataNodes, which enable data locality (putting the data close to where it is needed) for the data served by the RegionServers. HBase data is local when it is written, but when a region is moved, it is not local until compaction.

The NameNode maintains metadata information for all the physical data blocks that comprise the files.



**Regions**

HBase Tables are divided horizontally by row key range into “Regions.” A region contains all rows in the table between the region’s start key and end key. Regions are assigned to the nodes in the cluster, called “Region Servers,” and these serve data for reads and writes. A region server can serve about 1,000 regions.



**HBase HMaster**

Region assignment, DDL (create, delete tables) operations are handled by the HBase Master.

A master is responsible for:

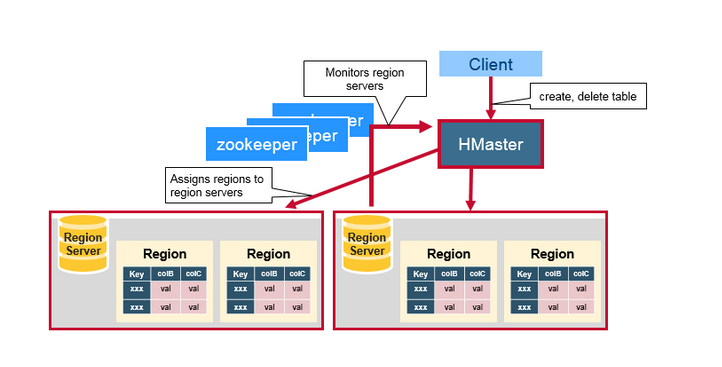
* Coordinating the region servers

- Assigning regions on startup , re-assigning regions for recovery or load balancing

- Monitoring all RegionServer instances in the cluster (listens for notifications from zookeeper)

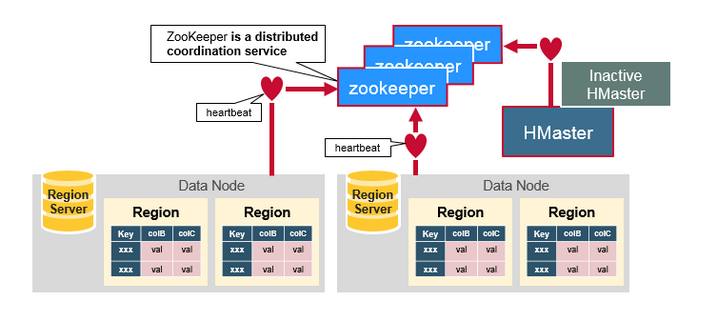
* Admin functions

- Interface for creating, deleting, updating tables



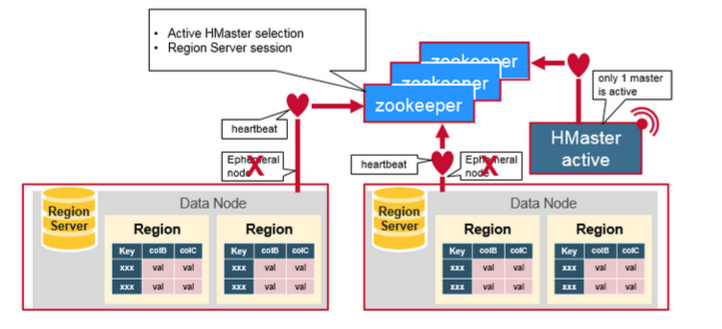
**ZooKeeper: The Coordinator**

HBase uses ZooKeeper as a distributed coordination service to maintain server state in the cluster. Zookeeper maintains which servers are alive and available, and provides server failure notification. Zookeeper uses consensus to guarantee common shared state. Note that there should be three or five machines for consensus.



**How the Components Work Together**

Zookeeper is used to coordinate shared state information for members of distributed systems. Region servers and the active HMaster connect with a session to ZooKeeper. The ZooKeeper maintains ephemeral nodes for active sessions via heartbeats.



Each Region Server creates an ephemeral node. The HMaster monitors these nodes to discover available region servers, and it also monitors these nodes for server failures. HMasters vie to create an ephemeral node. Zookeeper determines the first one and uses it to make sure that only one master is active. The active HMaster sends heartbeats to Zookeeper, and the inactive HMaster listens for notifications of the active HMaster failure.

If a region server or the active HMaster fails to send a heartbeat, the session is expired and the corresponding ephemeral node is deleted. Listeners for updates will be notified of the deleted nodes. The active HMaster listens for region servers, and will recover region servers on failure. The Inactive HMaster listens for active HMaster failure, and if an active HMaster fails, the inactive HMaster becomes active.

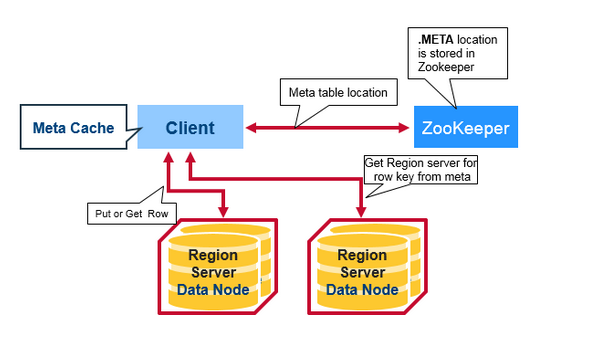
**HBase First Read or Write**

There is a special HBase Catalog table called the META table, which holds the location of the regions in the cluster. ZooKeeper stores the location of the META table.

This is what happens the first time a client reads or writes to HBase:

1. The client gets the Region server that hosts the META table from ZooKeeper.
2. The client will query the .META. server to get the region server corresponding to the row key it wants to access. The client caches this information along with the META table location.
3. It will get the Row from the corresponding Region Server.

For future reads, the client uses the cache to retrieve the META location and previously read row keys. Over time, it does not need to query the META table, unless there is a miss because a region has moved; then it will re-query and update the cache.

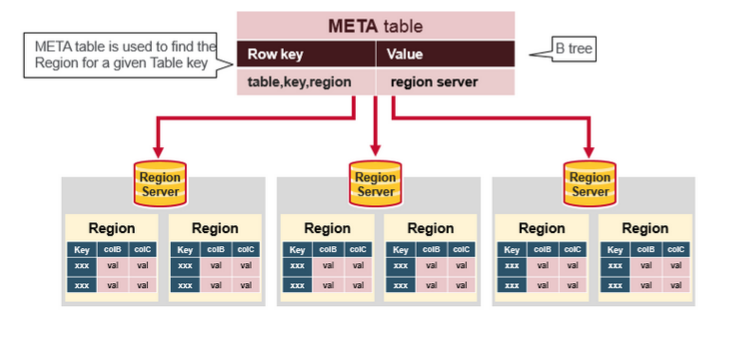


**HBase Meta Table**

* This META table is an HBase table that keeps a list of all regions in the system.
* The .META. table is like a b tree.
* The .META. table structure is as follows:

- Key: region start key,region id

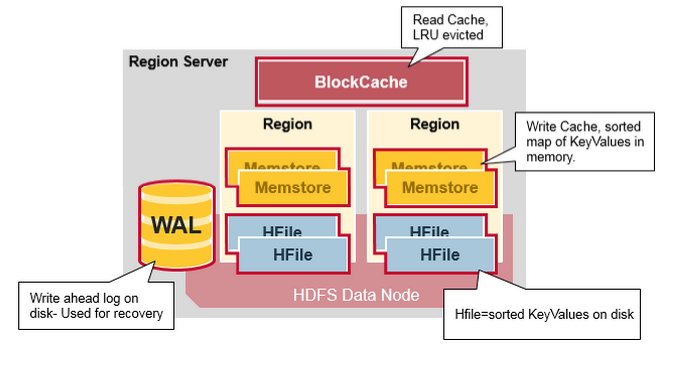
- Values: RegionServer



**Region Server Components**

A Region Server runs on an HDFS data node and has the following components:

* WAL: Write Ahead Log is a file on the distributed file system. The WAL is used to store new data that hasn't yet been persisted to permanent storage; it is used for recovery in the case of failure.
* BlockCache: is the read cache. It stores frequently read data in memory. Least Recently Used data is evicted when full.
* MemStore: is the write cache. It stores new data which has not yet been written to disk. It is sorted before writing to disk. There is one MemStore per column family per region.
* Hfiles store the rows as sorted KeyValues on disk.

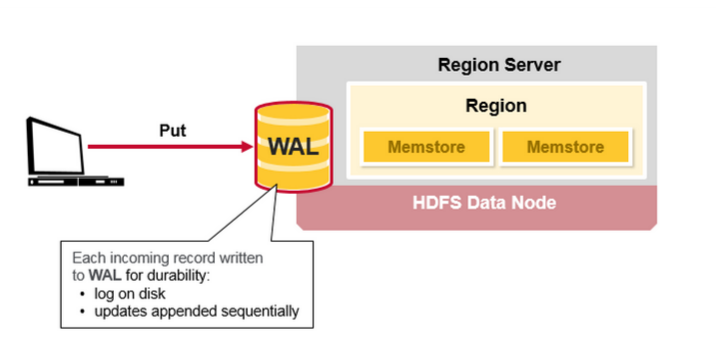


**HBase Write Steps (1)**

When the client issues a Put request, the first step is to write the data to the write-ahead log, the WAL:

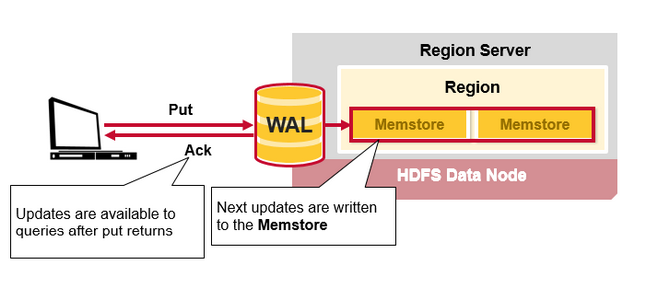
- Edits are appended to the end of the WAL file that is stored on disk.

- The WAL is used to recover not-yet-persisted data in case a server crashes.



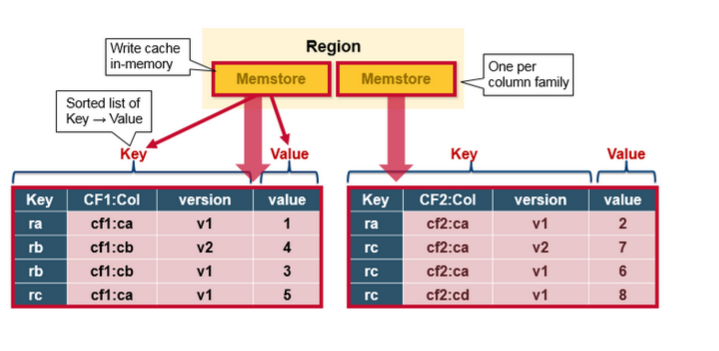
**HBase Write Steps (2)**

Once the data is written to the WAL, it is placed in the MemStore. Then, the put request acknowledgement returns to the client.



**HBase MemStore**

The MemStore stores updates in memory as sorted KeyValues, the same as it would be stored in an HFile. There is one MemStore per column family. The updates are sorted per column family.

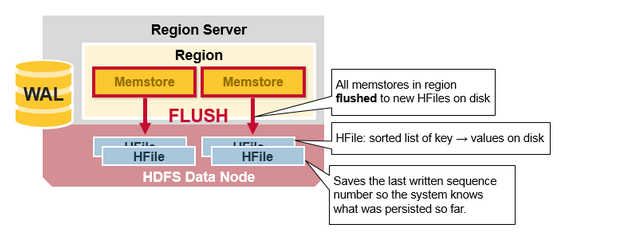


**HBase Region Flush**

When the MemStore accumulates enough data, the entire sorted set is written to a new HFile in HDFS. HBase uses multiple HFiles per column family, which contain the actual cells, or KeyValue instances. These files are created over time as KeyValue edits sorted in the MemStores are flushed as files to disk.

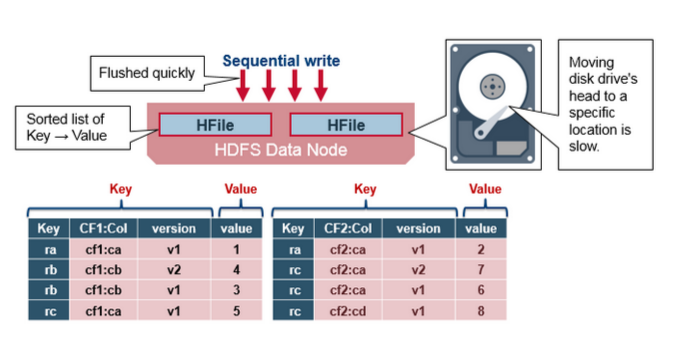
Note that this is one reason why there is a limit to the number of column families in HBase. There is one MemStore per CF; when one is full, they all flush. It also saves the last written sequence number so the system knows what was persisted so far.

The highest sequence number is stored as a meta field in each HFile, to reflect where persisting has ended and where to continue. On region startup, the sequence number is read, and the highest is used as the sequence number for new edits.



**HBase HFile**

Data is stored in an HFile which contains sorted key/values. When the MemStore accumulates enough data, the entire sorted KeyValue set is written to a new HFile in HDFS. This is a sequential write. It is very fast, as it avoids moving the disk drive head.

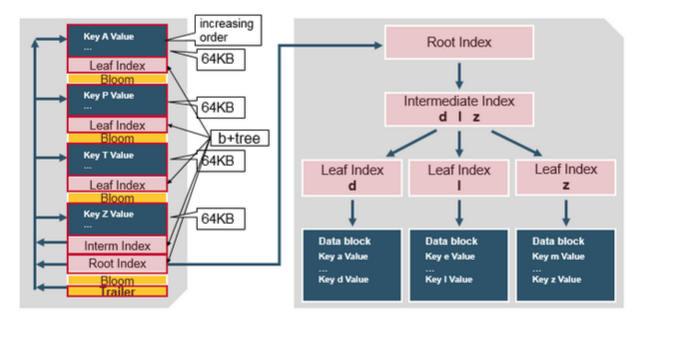


**HBase HFile Structure**

An HFile contains a multi-layered index which allows HBase to seek to the data without having to read the whole file. The multi-level index is like a b+tree:

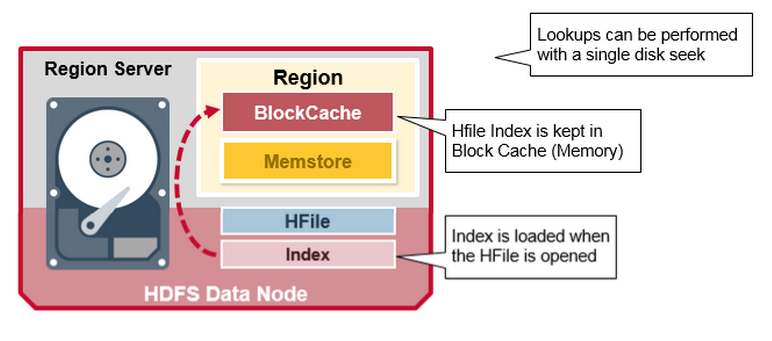
* Key value pairs are stored in increasing order
* Indexes point by row key to the key value data in 64KB “blocks”
* Each block has its own leaf-index
* The last key of each block is put in the intermediate index
* The root index points to the intermediate index

The trailer points to the meta blocks, and is written at the end of persisting the data to the file. The trailer also has information like bloom filters and time range info. Bloom filters help to skip files that do not contain a certain row key. The time range info is useful for skipping the file if it is not in the time range the read is looking for.



**HFile Index**

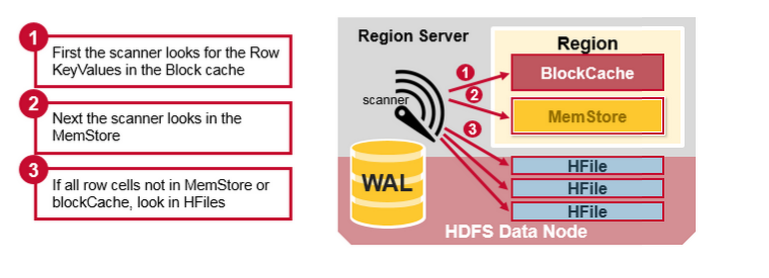
The index, which we just discussed, is loaded when the HFile is opened and kept in memory. This allows lookups to be performed with a single disk seek.



**HBase Read Merge**

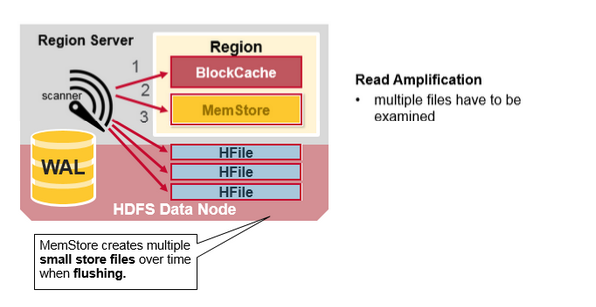
We have seen that the KeyValue cells corresponding to one row can be in multiple places, row cells already persisted are in Hfiles, recently updated cells are in the MemStore, and recently read cells are in the Block cache. So when you read a row, how does the system get the corresponding cells to return? A Read merges Key Values from the block cache, MemStore, and HFiles in the following steps:

1. First, the scanner looks for the Row cells in the Block cache - the read cache. Recently Read Key Values are cached here, and Least Recently Used are evicted when memory is needed.
2. Next, the scanner looks in the MemStore, the write cache in memory containing the most recent writes.
3. If the scanner does not find all of the row cells in the MemStore and Block Cache, then HBase will use the Block Cache indexes and bloom filters to load HFiles into memory, which may contain the target row cells.



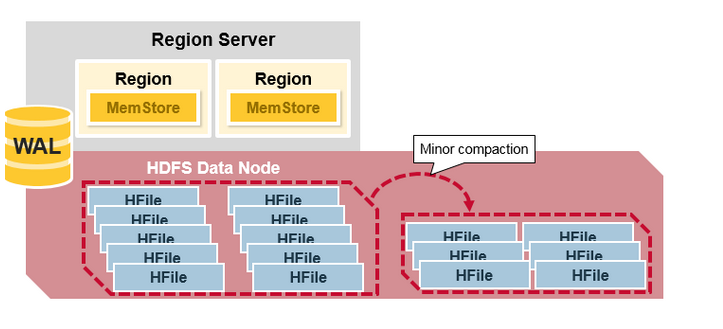
**HBase Read Merge**

As discussed earlier, there may be many HFiles per MemStore, which means for a read, multiple files may have to be examined, which can affect the performance. This is called read amplification.



**HBase Minor Compaction**

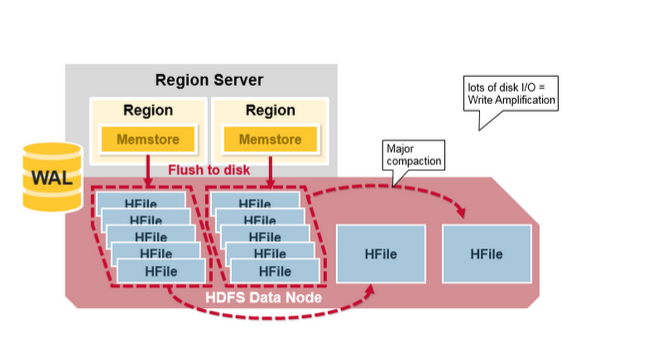
HBase will automatically pick some smaller HFiles and rewrite them into fewer bigger Hfiles. This process is called minor compaction. Minor compaction reduces the number of storage files by rewriting smaller files into fewer but larger ones, performing a merge sort.



**HBase Major Compaction**

Major compaction merges and rewrites all the HFiles in a region to one HFile per column family, and in the process, drops deleted or expired cells. This improves read performance; however, since major compaction rewrites all of the files, lots of disk I/O and network traffic might occur during the process. This is called write amplification.

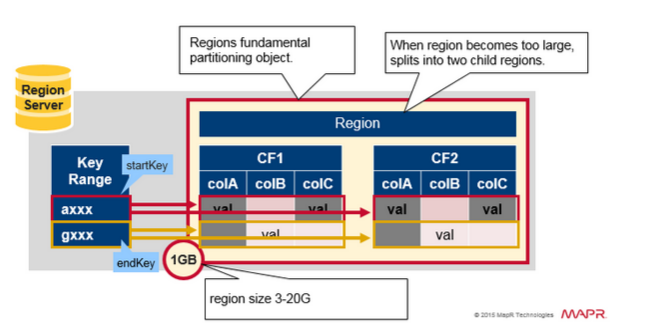
Major compactions can be scheduled to run automatically. Due to write amplification, major compactions are usually scheduled for weekends or evenings. Note that MapR-DB has made improvements and does not need to do compactions. A major compaction also makes any data files that were remote, due to server failure or load balancing, local to the region server.



**Region = Contiguous Keys**

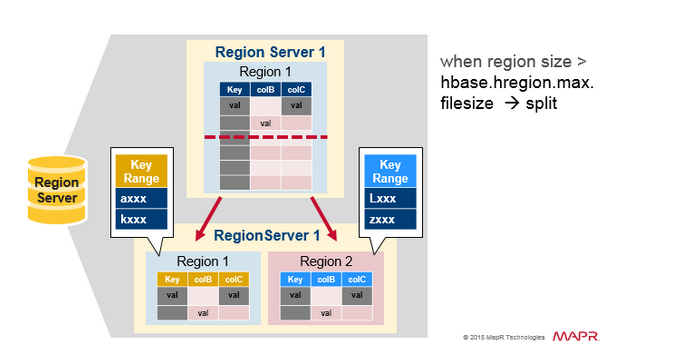
Let’s do a quick review of regions:

* A table can be divided horizontally into one or more regions. A region contains a contiguous, sorted range of rows between a start key and an end key
* Each region is 1GB in size (default)
* A region of a table is served to the client by a RegionServer
* A region server can serve about 1,000 regions (which may belong to the same table or different tables)



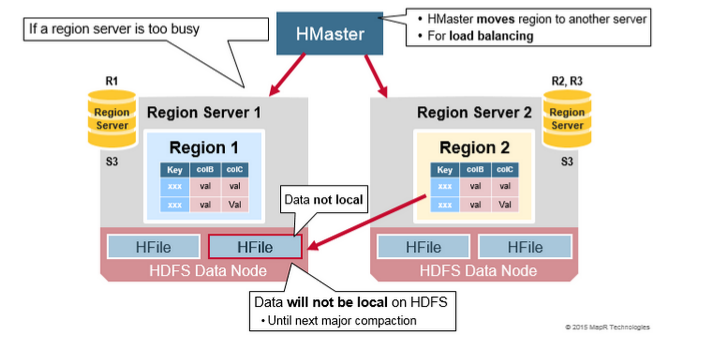
**Region Split**

Initially there is one region per table. When a region grows too large, it splits into two child regions. Both child regions, representing one-half of the original region, are opened in parallel on the same Region server, and then the split is reported to the HMaster. For load balancing reasons, the HMaster may schedule for new regions to be moved off to other servers.



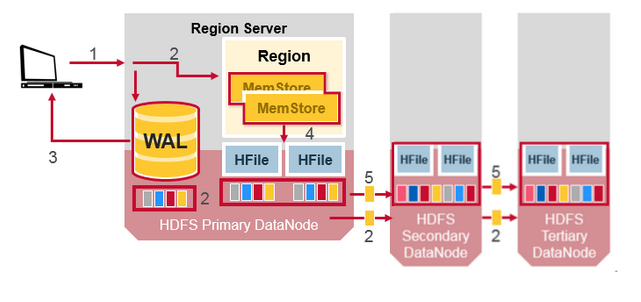
**Read Load Balancing**

Splitting happens initially on the same region server, but for load balancing reasons, the HMaster may schedule for new regions to be moved off to other servers. This results in the new Region server serving data from a remote HDFS node until a major compaction moves the data files to the Regions server’s local node. HBase data is local when it is written, but when a region is moved (for load balancing or recovery), it is not local until major compaction.



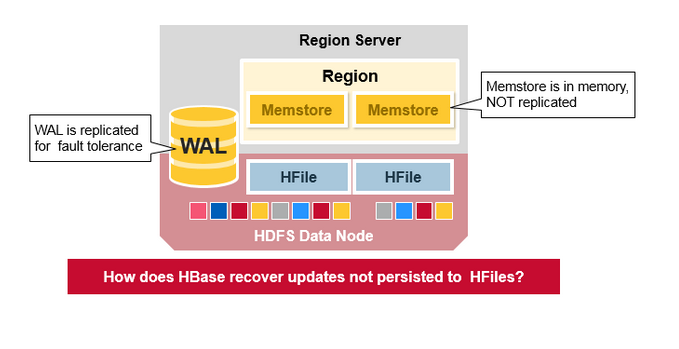
**HDFS Data Replication**

All writes and Reads are to/from the primary node. HDFS replicates the WAL and HFile blocks. HFile block replication happens automatically. HBase relies on HDFS to provide the data safety as it stores its files. When data is written in HDFS, one copy is written locally, and then it is replicated to a secondary node, and a third copy is written to a tertiary node.



**HDFS Data Replication (2)**

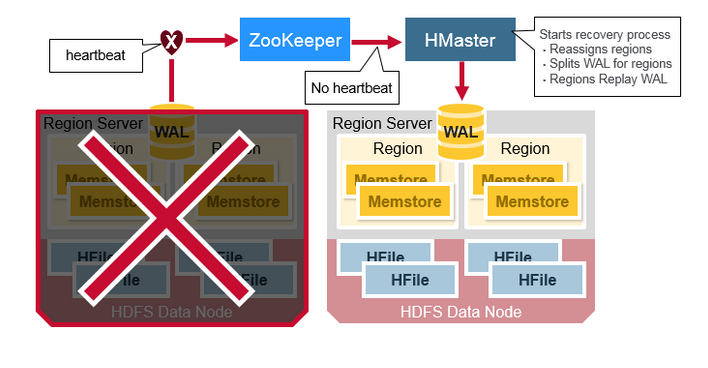
The WAL file and the Hfiles are persisted on disk and replicated, so how does HBase recover the MemStore updates not persisted to HFiles? See the next section for the answer.



**HBase Crash Recovery**

When a RegionServer fails, Crashed Regions are unavailable until detection and recovery steps have happened. Zookeeper will determine Node failure when it loses region server heart beats. The HMaster will then be notified that the Region Server has failed.

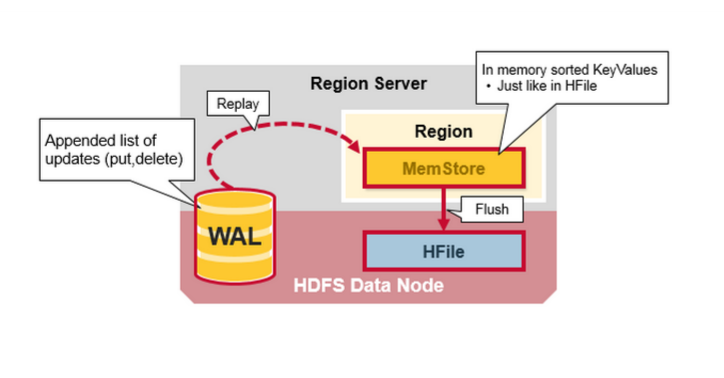
When the HMaster detects that a region server has crashed, the HMaster reassigns the regions from the crashed server to active Region servers. In order to recover the crashed region server’s memstore edits that were not flushed to disk. The HMaster splits the WAL belonging to the crashed region server into separate files and stores these file in the new region servers’ data nodes. Each Region Server then replays the WAL from the respective split WAL, to rebuild the memstore for that region.



**Data Recovery**

WAL files contain a list of edits, with one edit representing a single put or delete. Edits are written chronologically, so, for persistence, additions are appended to the end of the WAL file that is stored on disk.

What happens if there is a failure when the data is still in memory and not persisted to an HFile? The WAL is replayed. Replaying a WAL is done by reading the WAL, adding and sorting the contained edits to the current MemStore. At the end, the MemStore is flush to write changes to an HFile.



**Apache HBase Architecture Benefits**

HBase provides the following benefits:

* **Strong consistency model**

- When a write returns, all readers will see same value

* **Scales automatically**

- Regions split when data grows too large

- Uses HDFS to spread and replicate data

* **Built-in recovery**

- Using Write Ahead Log (similar to journaling on file system)

* **Integrated with Hadoop**

- MapReduce on HBase is straightforward

**Apache HBase Has Problems Too…**

* **Business continuity reliability:**

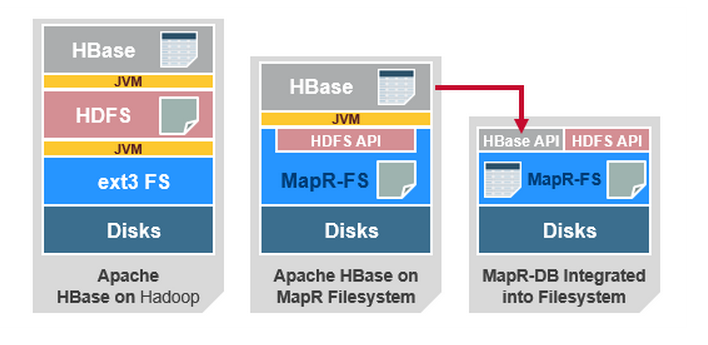
- WAL replay slow

- Slow complex crash recovery

- Major Compaction I/O storms

**MapR-DB with MapR-FS does not have these problems**

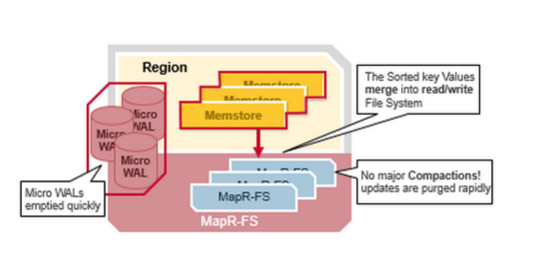
The diagram below compares the application stacks for Apache HBase on top of HDFS on the left, Apache HBase on top of MapR's read/write file system MapR-FS in the middle, and MapR-DB and MapR-FS in a Unified Storage Layer on the right.



MapR-DB exposes the same HBase API and the Data model for MapR-DB is the same as for Apache HBase. However the MapR-DB implementation integrates table storage into the MapR file system, eliminating all JVM layers and interacting directly with disks for both file and table storage.

MapR-DB offers many benefits over HBase, while maintaining the virtues of the HBase API and the idea of data being sorted according to primary key. MapR-DB provides operational benefits such as no compaction delays and automated region splits that do not impact the performance of the database. The tables in MapR-DB can also be isolated to certain machines in a cluster by utilizing the topology feature of MapR. The final differentiator is that MapR-DB is just plain fast, due primarily to the fact that it is tightly integrated into the MapR file system itself, rather than being layered on top of a distributed file system that is layered on top of a conventional file system.

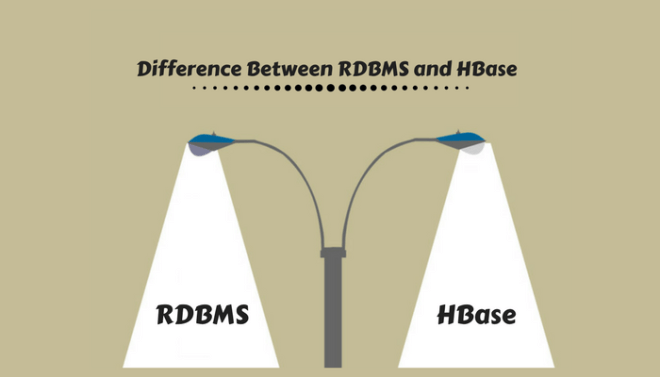
**Key differences between MapR-DB and Apache HBase**



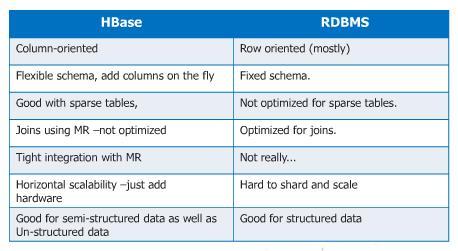
* Tables part of the MapR Read/Write File system
  + Guaranteed data locality
* Smarter load balancing
  + Uses container Replicas
* Smarter fail over
  + Uses container replicas
* Multiple small WALs
  + Faster recovery
* Memstore Flushes Merged into Read/Write File System

**HBase vs RDBMS**

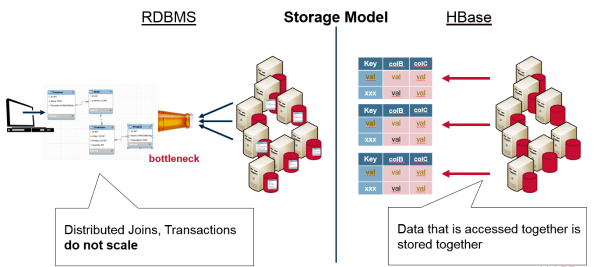
# **Difference between RDBMS and HBase**



Hadoop and RDBMS are varying concepts of processing, retrieving and storing the data or information. While Hadoop is an open-source Apache project, RDBMS stands for Relational Database Management System. Hadoop framework has been written in Java which makes it scalable and makes it able to support applications that call for high performance standards. Hadoop framework enables the storage of large amounts of data on files systems of multiple computers. Hadoop is configured to allow scalability from a single computer node to several thousands of nodes or independent workstations in a manner that the individual nodes utilize local computer storage CPU processing power and memory.



Database Management Systems focus on the data storage in table form which includes columns and rows. SQL is utilized to retrieve needed data which is stored in such tables. The RDBMS concept stores relationships between such tables in various forms so that one column of entries of a particular table will act as a reference for another table. Such column values are known as primary keys and foreign keys with the keys being used to reference other existing tables so that the appropriate data can be related and also be retrieved by combining such tables using SQL queries as required. The tables and relationships can be altered by integrating relevant tables using SQL queries.

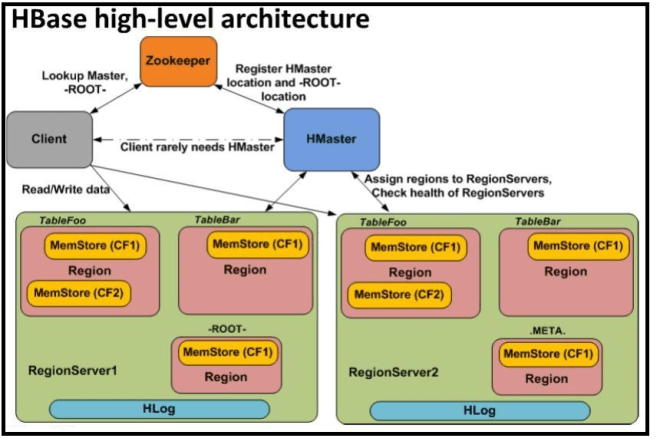


Source: [mapr.com](http://mapr.com/)

It is important to remember that Hadoop is not an actual database, although HBase and Impala can be considered as databases. Hadoop is just a file system known as Hadoop File System (HDFS) with built-in redundancy and parallelism. Traditional databases or RDBMS’ have “ACID” properties which is an acronym that stands for Atomicity, Consistency, Isolation and Durability. These properties are not found at all in Hadoop. For instance if one has to script code for taking money from one particular bank to deposit the same into another bank then, they will have to painstakingly code all scenarios that may occur such as what happens when money is taken out but a failure results before it is moved into another account.

However, Hadoop brings a lot to the table too, offering massive scaling capability in terms of processing power and storage at costs which are relatively much lower when compared to the sky high costs of an RDBMS. Hadoop also comes packed with amazing parallel processing capabilities, with one being able to run jobs in parallel to execute number crunching in large volumes.

When it comes to deciding the efficacy of traditional databases in terms if working with unstructured data, it is not so simple. Several applications that are created utilizing traditional RDBMS which use a lot of unstructured data or video files or PDFs seem to work well, as vouched for by market experts. Usually RDBMS is meant to handle large chunks of data in its cache for accelerated processing while simultaneously maintaining read consistency across sessions. Hadoop does a much better job of utilizing the memory cache for data processing without providing any other aspects such as read consistency. Hive SQL has always known to be several times slower than SQL which can be run in traditional databases. Users who go in thinking that SQL in Hive is speedier than in databases will be in for a huge let-down since it will not scale to any extent whatsoever for complicated analytics. While Hadoop excels at parallel processing problems such as finding a group of keywords in large sets of documents, RDBMS implementations will be much faster for data sets which are comparable.



It will come as no surprise to any tech enthusiast that the volumes of data being generated are explosive with seconds adding up to huge data volumes that need processing. This means that the traditional dates that were developed from a single CPU and RAM cache premise will not be able to support business needs of an enterprise in the future.  If one weighs the benefits of an old style reports with high consistencies with that of instant reports with reasonable or partially consistent aspects than it is better to go for on that is more current. The next step for evolution for both Hadoop and RDBMS would be to patch up both their short-coming to deliver a satisfactory customer usage experience.

**Why and when would you choose one over the other?**  
Hadoop is an excellent way to replace an organization’s tape back-up strategy with it acting as a great Super-fast Tapeless Backup.

Hadoop is great for when one has to sift through endless Gigabytes and Terabytes of data and does not know what to do with the data with one not knowing what to do with the data. Hadoop is very user-friendly since it can be used by people not equipped with the know-how to handle data. There is no workaround to the fact that a user will have to model his/her data, and any exception made in this regard will bring about fundamental problems of great scale.

Hadoop is custom-made for large quantities of structured data and unstructured data is usually rare such as email, or poorly designed log files, twitter feeds and www web pages. Such data is used when users have not put in the time to understand their data. RDBMS are used whenever a consumer wants to manage data, its relationships and when the user cares about data governance and security at large scale budgets and platforms used at such budgets.

Traditional RDBMS is utilized to handle relational data while Hadoop works well with structured as well as unstructured data, supporting multiple serialization and data formats such as Text, Json, Xml, Avro and more. The distribution of Hadoop is done from the ground up with the option of adding more nodes in order to boost capacity.

The only place where problems would arise would be where SQL databases are the only choice. If the data size and type allows for it, with the data type being relational, oone can go ahead with the RDBMS route since it is a time tested method and a matured tech.

When the data size or type is such that one is unable to save it in an RDBMS, then one should select Hadoop as their way out. The perfect example for this would be product catalogue. An automobile would have a much higher number of attributes than a TV which will make it tough to construct a new table as per the product type. Another suitable example is when dealing with machine generated data where the data size puts a large stress on traditional RDBMS – which serves as a classic case of Hadoop coming to the rescue. Another obvious example would be document indexing.